ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.17, No.12, December 2006, pp.2403−2415
DOI: 10.1360/jos172403

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

多模式匹配算法及硬件实现*

李伟男[1,2+], 鄂跃鹏[1,2], 葛敬国[1], 钱华林[1]

[1](中国科学院 计算机网络信息中心,北京 100080)

[2](中国科学院 研究生院,北京 100049)

Multi-Pattern Matching Algorithms and Hardware Based Implementation

LI Wei-Nan[1,2+], E Yue-Peng[1,2], GE Jing-Guo[1], QIAN Hua-Lin[1]

[1](Computer Network Information Center, The Chinese Academy of Sciences, Beijing 100080, China)

[2](Graduate School, The Chinese Academy of Sciences, Beijing 100049, China)

+ Corresponding author: Phn: +86-10-58812364, Fax: +86-10-58812306, E-mail: wnli@cnic.cn, http://www.cnic.cn

Li WN, E YP, Ge JG, Qian HL. Multi-Pattern matching algorithms and hardware based implementation. *Journal of Software*, 2006,17(12):2403−2415. http://www.jos.org.cn/1000-9825/17/2403.htm

Abstract: This paper surveys the algorithms and hardware implementations of the multi-pattern matching. Firstly, two commonly used multi-pattern algorithms, Aho-Corasick's automata based algorithm and Wu-Manber's hash based suffix matching with skipping algorithm are introduced. And then, some improving ways are referred. Next, time and space complexity of these algorithms are analyzed, and the experimental results show their performances. Further, several hardware based implementations are taken as examples to demonstrate the general methods and strategies for the implementation on hardware. The developing trend is predicted in the end.

Key words: multi-pattern matching; Aho-Corasick algorithm; finite state automata; Wu-Manber algorithm; FPGA (field programmable gate array); TCAM (ternary content addressable memory); bloom filter

摘 要: 介绍了多模式匹配的算法和硬件实现方法.首先介绍了两种常用的多模式匹配算法——Aho-Corasick 基于自动机的算法和 Wu-Manber 基于 hash 的后缀匹配加移位跳跃的算法以及相关的改进算法.并通过实验对各种多模式匹配算法的时空复杂度进行了分析比较.通过几个硬件实现的实例介绍了多模式匹配的硬件实现方法及策略.最后对多模式匹配的发展趋势进行了展望.

关键词: 多模式匹配;Aho-Corasick 算法;有限状态自动机;Wu-Manber 算法;FPGA(现场可编程门阵列);TCAM(三态内容寻址存储器);bloom filter

中图法分类号: TP301 文献标识码: A

模式匹配问题是计算机科学中的一个基本问题,其研究内容在信息检索、模式识别等众多领域均有重要价值,在拼写检查、语言翻译、数据压缩、搜索引擎、入侵检测、内容过滤、计算机病毒特征码匹配以及基因序列比较等应用中起着重要的作用.模式匹配按照匹配模式的数目分为单模式匹配和多模式匹配两类.最初的单

* Supported by the National Natural Science Foundation of China under Grant No.90412011 (国家自然科学基金); the Special Foundation of President of the Chinese Academy of Sciences under Grant No.KGCX2-YW-106 (中国科学院院长基金)

Received 2006-05-16; Accepted 2006-08-18

模式匹配 KMP 算法、Commentz-Walter 算法以及 Boyer-Moore 算法等为后来多模式的发展提供了一些借鉴与启发.因为多模式匹配通过一次扫描可以找到多个模式的所有出现,较之于单模式匹配实现复杂,应用范围也更广泛,所以本文主要围绕多模式匹配来展开.

1 概 述

设集合 $P=\{p_1,p_2,\ldots,p_k\}$是一组模式的集合,每个模式 p_i 由来源于字符集合Σ的字符串组成.给定长度为 n 的文本串 $T=T[1\ldots n]$,文本串的每个字符也来源于字符集集合Σ.多模式匹配就是在文本 T 中找出模式集合 P 中的每个模式的所有出现.

Aho 和 Corasick 于 1975 年最早提出了基于自动机的多模式匹配算法[1],随后,围绕这一算法,一些改进的算法被提了出来,比如压缩存储空间基于位图的算法[2]、反向构建自动机引入跳跃的启发式方法[3]等.而 Wu 和 Manber 于 1994 年提出了采用 hash 方法的基于后缀匹配方法,同时采用移位来加速比较[4,5].这两种方法是当前主流的软件实现的多模式匹配算法,本文主要介绍这两种基本算法及其相关的改进.随着各种新的应用的出现,基于软件的实现已经不能充分满足对匹配速度日益增长的需求,因此,基于硬件的多模式匹配的实现逐渐出现,并成为当前的研究热点.本文后面以 3 个具体的实现为例,介绍了基于硬件的实现方法和策略.

本文第 2 节首先介绍基本的 Aho-Corasick 算法,而后依次介绍 3 种基于它的改进算法.第 3 节介绍 Wu-Manber 算法.第 4 节给出实验数据,比较上述算法的性能.第 5 节给出多模式匹配的一些硬件实现方法策略以及具体的实现.最后对多模式匹配的发展趋势进行了展望.

2 Aho-Corasick 算法

2.1 基本的Aho-Corasick算法

Aho-Corasick 算法[1]是基于有穷状态自动机的,在进行匹配之前,先对模式串集合进行预处理,构建树型有限状态自动机 FSA(finite state automata),然后,依据该 FSA,只需对文本串 T 扫描一次就可以找出与其匹配的所有模式串.预处理生成 3 个函数:goto(转移)函数、failure(失效)函数和 output(输出)函数.转移函数 goto 表明,在当前状态下读入下一个待比较文本的字符后到达的下一个状态.失效函数 failure 用来指明在某个状态下,当读入的字符不匹配时应转移到的下一个状态.输出函数 output 的作用是,在匹配过程中,当出现匹配时输出匹配到的模式.

Aho-Corasick 算法的匹配过程是:从初始状态 0 出发,每次取出文本串中的一个字符,根据当前的状态和扫描到的字符,利用 goto 或 failure 函数进入下一状态.当某个状态的 output 函数不为空时,表明达到该状态时找到了匹配的模式,于是输出其值.

下面首先介绍 3 个预处理函数的构建.为了构造 goto 函数,首先要根据模式构建 goto 图,设置一个初始状态 0,然后依次扫描各个模式,根据模式逐步为这个图添加新的边和顶点,构造出自动机:从状态 0 出发,由当前状态和读到的下一个字符决定下一状态.如果有从当前状态出发并标注该字符的矢线,则将矢线所指的状态赋为当前状态;否则,添加一个标号比已有状态标号大 1 的新状态,并用一条矢线从当前状态指向新加入的状态,并将新加入的状态赋为当前状态;所有模式串处理完毕后,再画一条 0 状态的自返矢线,标注的字符为不能从 0 开始的字符集.这样,每个模式都可以由一条从初始状态出发的路径标识出来.对模式串集合{she,he,hers,his}构建的 goto 图如图 1 所示.

失效函数 f 是逐层构造的,设某个状态的层深度为初始状态到该状态的最短路径长度.令第 1 层状态的 failure 函数值为 0,如 $f(1)=f(3)=0$;对于非第 1 层状态 s,若其父状态为 r,即存在字符 a 使 $g(r,a)=s$(这里,s 的层深度比 r 少 1),则 $f(s)=g(f(s^*),a)$,其中,状态 s^*为追溯 s 的祖先状态得到的第 1 个使 $g(f(s^*),a)$存在的状态,如 $f(4)=g(f(3),h)=g(0,h)=1$,$f(5)=g(f(4),e)=g(1,e)=2$.

输出函数 output 的作用是在匹配过程中输出匹配到的模式串.output 的构造分两步:第 1 步是在构造转移函

数 g 时,每处理完一个模式串,则将该模式串加入到当前状态 s 的输出函数中,如 $output(2)=\{he\}$,$output(5)=\{she\}$.第 2 步是在构造失效函数 f 时,若 $f(s)=s'$,则 $output(s)=output(s)\cup output(s')$,如 $output(5)=output(5)\cup output(2)=\{she, he\}$.

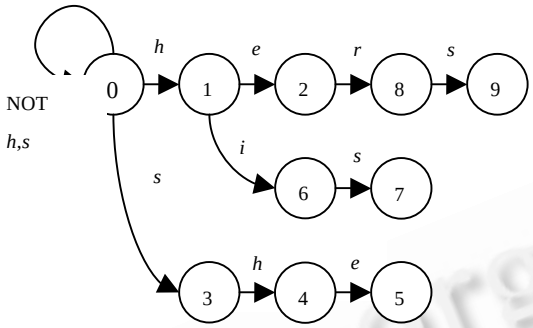

Fig.1 The goto graph for the Aho-Corasic algorithm

图 1 依据 Aho-Corasick 算法构建的 goto 图

构造完 3 个函数以后,就可以依次扫描文本,逐个读取输入字符.从状态 0 开始,根据当前状态和输入的字符,采用 goto 和 failure 函数转移到下一个状态,当到达状态的 output 函数不空时输出匹配模式.

Aho-Corasick 算法模式匹配的时间复杂度是 $O(n)$,而且与模式集中模式串的个数和每个模式串的长度无关.无论模式串 P 是否出现在 T 中,T 中的每个字符都必须输入状态机中,所以,无论是在最好情况还是最坏情况下,Aho-Corasick 算法模式匹配的时间复杂度都是 $O(n)$.包括预处理时间在内,Aho-Corasick 算法总时间复杂度是 $O(M+n)$,其中,M 为所有模式串的长度总和.

2.2 基于Aho-Corasick算法的改进方法

2.2.1 去掉 failure 函数的改进 Aho-Corasick 算法

基于基本 Aho-Corasick 算法,通过修改转移函数,可以去掉 failure 函数,将其合并到 goto 函数中.这样,在对文本进行匹配的过程中,对于每个输入字符仅进行一次状态转移,而不必在调用 goto 函数失败的情况下,再去通过 failure 函数进行状态转移.对于长度为 d 的字符串,最坏情况可能进行 d–1 次 failure 转移.文献[1]在给出基本 A-C 实现后,提出了改进的方式,通过合并 failure 函数和 goto 函数,构建 next_move 函数δ.如果 $g(s,a)$!=failure,那么 $g(s,a)=\delta(s,a)$;否则,$\delta(s,a)=\delta(f(s),a)$.这样,整个转移过程就只根据$\delta$函数进行.利用这个确定的有穷自动机(DFA)可以减少状态转移的次数,使得效率得到一定程度的提高.

2.2.2 位图压缩的 Aho-Corasick 算法

对于 goto 函数的存储,基本 A-C 算法采用二维数组,然而,实际上,这个数组的很多元素是空的,因为每个状态的下一个有效转移状态通常是很少的几个,这样就浪费了大量的存储空间.假设字符集大小为 256,为了节约空间,仅仅为每个状态保存指向第一个下一个有效状态的指针以及 256 比特的位图,用以指示读入哪些字符可以跳转到下一个有效状态,哪些导致失效.这里,每一个比特位对应一个输入的字符,如果输入相应的字符能够跳转到下一个有效状态,那么该位置“1”,否则置“0”.而对于所有指向有效的下一个状态的指针,采用连续的地址空间存储,这样,只要找到了第 1 个有效状态,再根据位图信息就可以找出对应的有效状态相对于第 1 个有效状态的偏移,于是就能够找到对应状态的指针,这就是基于位图压缩的 A-C 算法[2].图 2 给出了表示每个状态的存储结构.

failptr 指向当前匹配失效情况下应该转移到的下一个状态,功能上相当于原来的 failure 函数.

patternptr 指向到达当前状态时匹配出的模式串,功能上相当于原来的 output 函数.

nextptr 指向第 1 个下一个有效状态 1st *valid state*(所有的有效状态:1st *valid state*,2nd *valid state*,3rd *valid state*,在内存连续空间中存储).bitmap 位图每位标识一个字符,指示输入该字符能否达到有效状态.

对于每次查询,首先根据位图判断当前状态下,读入的字符是否能够产生有效的状态转移,这里是通过测试

位图中该字符对应的比特位是否为“1”来实现的.如果为“0”表明失效,则通过 *failptr* 指针实现失效状态下的转移;否则,计算该字符对应的转移状态前面还有多少个有效状态,这里就是计算位图中该字符对应的比特位之前还有多少个“1”,从而获得偏移量.由于有效状态是连续存储的,通过将偏移量加上第 1 个有效状态的地址,就能够找到当读入当前字符时,应该转到的下一个有效状态.

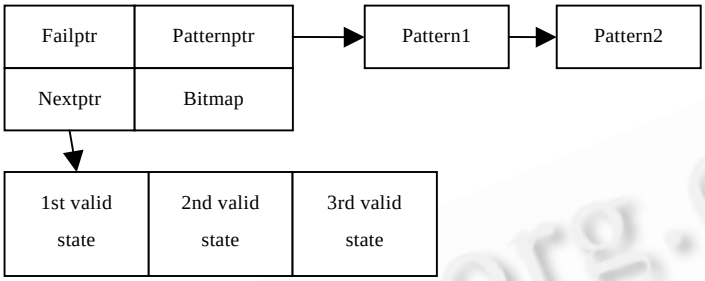

Fig.2 The storage structure of each state in the bitmap based A-C algorithm

图 2 位图压缩 A-C 算法中的每个状态的存储结构

该算法压缩了存储空间,在指针长度为 32bit 的机器上,当字符集大小为 256 时,那么,位图占 32 字节,每个状态存储需要 44 字节的空间;而基本的 Aho-Corasick 算法每个状态需要存储对于所有字符所转移到的下一个状态,则需要 4×256 字节的存储空间.即便考虑到位图压缩算法中为存储所有下一个有效状态提供的额外开销,存储空间的压缩也是相当可观的.但是,计算状态转移的过程引入了额外的开销,包括对于位图的测试以及计算偏移.

2.2.3 王方法

王永成等人[3]通过构建 goto 函数以及 skip 函数协助实现模式匹配.该算法对于模式采取从右向左进行比较,在一般情况下,该算法不需要匹配目标文本串中的每个字符,并充分利用了匹配过程中本次匹配不成功的信息,采用“坏字符”启发式规则,跳过尽可能多的字符,加快处理速度.

下面介绍两个主要的函数 goto 和 skip 的构造.对于 goto 函数的构造,与基本的 Aho-Corasick 算法类似,也是设初始状态为 0,因为算法是从右向左进行模式比较,进行逆向匹配,所以采用从后向前扫描模式串,构建逆向自动机,图 3 给出了对于模式串集合为{her,where,redo}构建出的 goto 图.

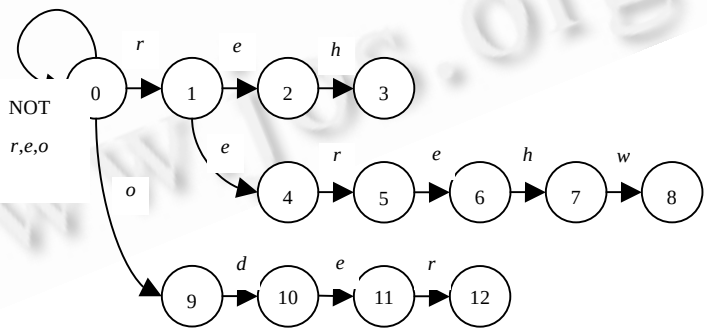

Fig.3 The reversely constructed goto graph

图 3 反向构建的 goto 图

skip 函数是用来依据当前字符判断向后可以跳过扫描的字符数目.这里,假设模式集合中最短模式的长度为 min*len*,那么

$$skip(char)=\begin{cases}\min\{j+1 \mid p_i[m_i-j]=char,\ 0\le j<\min len且1\le i\le k\}, & char出现在模式集合P\\ \min len+1, & char不出现在模式集合P中\end{cases}.$$

skip(*char*)的值是字符 char 在模式串中最右出现的位置到该模式串串尾的距离加 1.之所以加 1,是因为在匹配过程中计算 *skip* 时考虑了文本串中对齐模式串右边界的再右边一个字符,即计算了 *skip*(*T*[*i*+1]),这里,*i* 指向模式串的右边界所在位置;但是 *skip* 不能超过 min*len*+1,如果超过则有可能漏过最短的模式串.

匹配过程是从文本的 min*len* 位开始扫描,从右向左比较,并令最初状态为 0.一般地,设某一次匹配从 *j*=0,*T*[*i*–*j*]开始,按照 goto 函数向左匹配,如果匹配成功,则令 *j*=*j*+1,匹配下一个 *T*[*i*–*j*];如果匹配失败,则令 *i*=*i*+*skip*(*T*[*i*+1]),重新开始一次匹配.在匹配过程中,如果已经匹配成功某个模式串,则输出模式串在文本串中的位置.

该算法预处理的时间复杂度为 $O(M+|c|)$,*M* 是模式串总长度,*c* 表示字符集合大小.而查找的时间复杂度与待查询的文本长度、各字符在文本中出现的概率、最短模式长度有关.最优情况是 $O(n/(\min len+1))$,最坏情况是 $O(n\times \max len)$.可以分析得出:最短模式越短,*skip* 函数的值就会很小,于是接近于每次移动一个字符,性能不可能有很大的提升.模式串中相同字符出现的概率越小,或者是相同字符间隔较大,则能提升 *skip* 值,在匹配过程中跳过较大范围,对于性能有积极的影响.

3 Wu-Manber 算法

Wu-Manber 算法采用了跳跃不可能匹配的字符策略和 hash 散列的方法,加速匹配的进行[4].该方法需要对所有模式进行预处理,构建 SHIFT,HASH 和 PREFIX 这 3 个表,便于后续处理.SHIFT 表用于在扫描文本串的时候,根据读入字符串决定可以跳过的字符数,如果相应的跳跃值为 0,则说明可能产生匹配,就要用到 HASH 表和 PREFIX 表进一步判断,以决定有哪些匹配候选模式,并验证究竟是哪个或者哪些候选模式完全匹配.下面首先介绍预处理过程.

假设模式集合 *P* 中最短的模式长度为 *m*,那么,后续讨论仅仅考虑所有模式的前 *m* 个字符组成的模式串,即要求所有匹配的模式长度相等.为了加快比较速度,对长为 *m* 的串进行分组,以 *B* 个长度的字符串为基本单位,每次比较长度为 *B* 的子串.对于 *B* 的选取,原文给出了指导公式计算出一个合适的 *B* 值:$B=\log_c 2M$.这里,$M=k\times m$,*k* 是模式的数目;而 *c* 表示字符集的大小即 $c=|\Sigma|$.

设 $X=x_1\ldots x_B$ 为 *T* 中的待比较的长度为 *B* 的子串,通过 hash 函数映射得到一个索引值 index,以该索引值作为偏移得到 SHIFT 表中的值,该值决定读到当前子串 *X* 后可以跳过的位数.假设 *X* 映射到 SHIFT 表的入口为 index 的表项,即 *index*=*hash*(*X*):

$$SHIFT[index]=\begin{cases} m-B+1, & \text{if } X \text{ doesn't appear in any pattern} \\ \min\{m-j \mid X[k]=P_i[j-B+k],\ \forall k,\ 1\le k\le B\}, & \text{if } X \text{ appears in some patterns} \end{cases}.$$

这样,就需要将每个模式中长度为 *B* 的子串 $a_{j-B+1}\ldots a_j$ 映射到 SHIFT 表中去,这里的映射函数是和上面相同的 hash 函数.

设当前比较的文本字串 *X* 的 hash 值为 *h*,如果 *SHIFT*[*h*]=0,说明可能产生了匹配,那么需要进一步进行判断.于是,用该 *h* 值作为索引,查 HASH 表找到 *HASH*[*h*],它存储的是指针,指向两个单独的表:一个是模式链表,另一个是 PREFIX 表.模式链表中存放的是后 *B* 个字符的 hash 值同为 *h* 的所有模式.PREFIX 表存储的是模式链表中每个模式的前缀 hash 值,它有利于进一步减少实际比较次数,因为往往有不少模式具有相同后缀,也就有相同的 hash 值,这样,它们都指向 HASH 表的同一表项,从而需要逐一比较每个可能的模式,增加了比较负担,通过引入前缀 hash 值的比较能够加速处理.对于待比较长度为 *m* 的串,如果其长度为 *B*′的前缀与模式的前缀的 hash 值也相同,则再将相应的文本串与符合的模式逐一进行比较,最终判定是否完全匹配.

预处理过程结束,完成 3 个表的构建.下面讨论扫描文本进行比较匹配的过程.匹配从文本的第 *m* 个字符开始,文本的扫描从左向右;对模式的匹配是从模式的后面向前进行的,即从右向左.每次扫描 *B* 个字符 $t_{m-B+1}\ldots t_m$,按如下步骤进行:

1) 计算这 *B* 个字符的 hash 值,得到 *h*;
2) 查 SHIFT 表找到 *SHIFT*[*h*]:如果大于 0,则根据这个值向后移动文本相应的长度,并转到 1);否则继续;
3) 计算当前指针往左的 *m* 个字符串长度为 *B*′的前缀 hash 值;

4) 查 HASH 表,找到 *HASH*[*h*]的指针 *p*,遍历模式链表,找到前缀 hash 值也相同的模式串;再将文本串和模式串逐一比较,判断是否匹配.如果匹配,则输出匹配模式串,并将文本向后移动一位,转 1),直到文本结束.

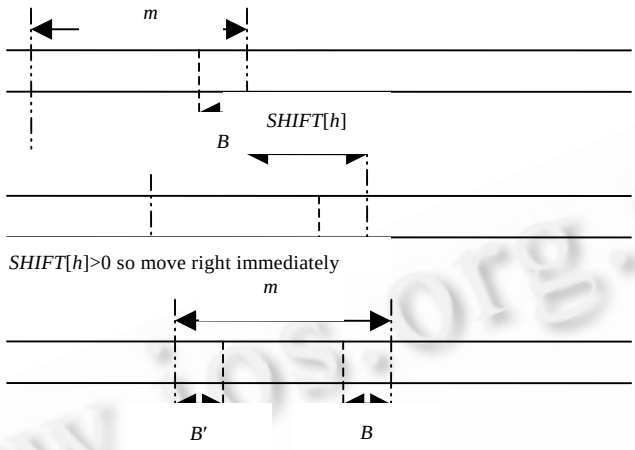

Fig.4 Wu-Manber algorithm searching for match progress

图 4 Wu-Manber 算法的扫描匹配

Wu-Manber 算法的时间复杂度平均情况是 $O(BN/m)$.B 是块字符的长度,而 N 是文本的长度,m 是模式的最短长度.该算法对最短模式长度敏感,SHIFT 函数的最大值受最短模式长度的限制,如果最短模式长度很短,则移位的值不可能很大,因此对匹配过程的加速有限.

4 实现的分析以及实验数据比较

4.1 预处理复杂度分析

上述这些算法都要进行预处理,生成相关的函数和表,可以在匹配之前独立完成.尽管它们的执行效率不影响匹配的速度,但它们也会占用一些资源,因此,我们先讨论一下各种算法的预处理的复杂度.对于基本的 Aho-Corasick 算法,生成 goto 函数需要扫描所有模式的每个字符,所以,时间复杂度随着所有模式的总长度线性增加;而 failure 函数的生成时间也与所有模式的总长度成正比;output 函数在 goto 和 failure 函数中各生成了一部分,如果用链接表存放匹配的模式,那么,failure 函数添加匹配模式时只需要常数时间,因此,整个 output 函数的构造时间也是由所有模式的总长度决定的.而去掉了 failure 函数的 A-C 算法,基于位图压缩的 A-C 算法都需要利用基本 A-C 算法生成的 3 个函数,时间复杂度与基本 A-C 算法相同.Wang 方法逆向构建状态机,其 goto 函数的时间复杂度是与基本 A-C 算法相同的,但是它没有 failure 函数,取而代之的是 skip 函数,而 skip 函数的构造需要扫描所有的模式,从而获得所有字符的 skip 值,因此,其时间复杂度随着所有模式的总长度而呈线性增加.对于 Wu-Manber 算法,SHIFT 表的构建需要扫描模式的每个 B 块,因而需要计算 hash 值获得表的入口偏移,并判断读入该块时可以产生的移位数目,每个模式需要计算 $m–B+1$ 次,设有 k 个模式,则总共需要计算$(m–B+1)\times k$ 次;HASH 表里的模式链表每个表项放的是后 B 位 hash 值相同的模式,这个 hash 计算可以合并到上面 SHIFT 表的构建中,而每个模式的前缀 B'的计算,如果令 $B'=B$,则也可以在计算 SHIFT 表的同时获得每个模式的前缀,这样,整个预处理的时间复杂度受模式数目、最短模式长度和选取的每次匹配块大小的影响.

下面讨论空间的复杂度,对于基本 Aho-Corasick 算法,goto 函数的存放可以是一个由状态和输入字符构成的二维数组,但是当状态集和字符集很大的时候,因为有效转移状态有限,二维数组存储会浪费很多存储空间存

放失效标记,所以,可以把非失效状态的转移用线性表存储起来,能够有效节省存储空间.而折衷的方案是:将频繁出现的状态(比如状态 0)存放在能通过输入字符和当前状态直接索引到的表中,以获取对这样的状态转换的常数访问时间,而将其他出现不频繁状态的状态转换放到线性表中.Failure 函数的存放可以采取一维数组,这样获取每个状态的 failure 值的时间是常数.状态对应输出模式通常用链表连接起来,通常情况下,一个状态最多只有一个输出,但最坏情况是某个状态的输出为所有的模式.对于 output 函数,所需要的总存储空间通常就是所有模式的存储空间.去掉了 failure 函数的 A-C 算法,仅仅在 goto 函数上与基本的算法有区别,由于每个状态对于每个输入都有下一个有效状态,所以采用由状态和输入字符构成的二维数组存放其改进后的 goto 函数.而 Wang 方法和基本 A-C 算法的 goto 函数存储方式基本相同,skip 表采用一维数组,大小为字符集的大小.基于位图压缩的 A-C 算法,前文已经分析过了存储格式、每个状态包含失效时的下一个状态指针、第 1 个有效状态指针、状态位图以及对应模式的指针.Wu-Manber 中的 SHIFT 表和 HASH 表如果采用无冲突的 hash 生成表项索引,均至少需要$|\Sigma|^B$个表项,而 HASH 表项指向所有的 k 个模式,这些模式的存储空间和所有模式的总长度成正比,同时,还有每个模式的前缀长度需要存储在 PREFIX 表中.

4.2 算法的实现和结果分析

对上述的这些算法进行实验测试,选取《人民日报》2003 年前 4 个月的内容共计 21MB 的文本数据,在其中查找多个模式串,找到每个模式在文本中的所有出现.对于遇到的匹配模式,将对应的计数器加 1,除此外不再做任何其他处理.分析模式长度变化和模式集合大小变化的情况下,各种算法的执行时间.预处理阶段在匹配之前完成,未包含在执行时间的统计中,统计只包含文本扫描匹配和计数的时间.设输入字符集合是值为 0~255 的 ASCII 字符.对于每一个汉字,将其识别为两个 ASCII 字符.实验运行于 Linux 平台,内核版本 2.4.20,CPU 为 Intel Pentium 4,2.8GHz,内存容量 512MB,所有算法均采用 C 来实现.以下用 AC 代表基本的 Aho-Corasick 算法,AC-Nofailure 代表去掉了 failure 函数的 Aho-Corasick 算法,AC-BM 代表基于位图压缩的 Aho-Corasick 算法,WANG 代表王永成等人的算法,WM 代表 Wu-Manber 算法.

4.2.1 基于位图压缩的 A-C 算法实现的优化

对于基于位图压缩的 A-C 算法,我们这里采用优化策略来提高匹配的速度.根据前面的介绍可知:由于 A-C 算法自身的特性,初始的 0 状态对于每个输入字符、转移函数 goto 都有有效输出——要么跳转到下一个状态,要么返回到自身.这样,在位图压缩的 A-C 算法中, 0 状态情况下,每个读入字符都能产生有效转移,从而不用测试位图字段中相应位是否置“1”,也不用计算当前状态前面还有多少个有效状态,而可以直接依据输入的字符对应的 ASCII 码作为偏移,再加上第 1 个有效状态指向的地址,就能找到下一个状态的指针,从而减少计算开销.因为 0 状态在匹配过程中是出现得非常频繁的状态,所以,提高 0 状态下状态转换的性能能够在很大程度上提高整体匹配性能.表 1 给出了待匹配模式数目为 10 的情况下,优化前后的位图压缩的 A-C 算法性能的比较.由此可以看出,优化以后,性能提升非常显著.

Table 1 Execution time of the basic bitmap A-C and optimized bitmap A-C (s)

表 1 优化前后位图压缩的 A-C 算法执行时间 (秒)

AC_BM	34.12
Opt_AC_BM	1.95

4.2.2 模式数目和长度变化对算法的影响

对于模式数目不同的情况,表 2 给出了各种算法的性能比较.由于对遇到的每个匹配我们都将相应的计数器加 1,所以应该考虑模式数目的增加会带来相应的开销.AC 算法在匹配失效时会执行 failure 函数向前返回,多次执行 failure 会带来性能的下降.而去掉了 failure 函数的 AC 算法,没有了 failure 函数的执行,读入每个输入都能转移到下一个有效状态,匹配的执行时间只与文本长度有关,而与模式数目和长度没有关系.Wang 方法由于引入了启发式思想,不必扫描文本中的每个字符,能够跳过一些不可能匹配的字符,因此在性能上比其他方法都优越.Wu-Manber 方法也引入了启发式方法,跳过不可能匹配的字符,但是由于用到了 hash 函数,增加了一定

的计算复杂性,而且即便 hash 值相同,还要进一步对模式逐字比较进行完全匹配,因而带来了一定的开销.

Table 2 Execution time of all the multi-pattern matching algorithms (s)

表 2 算法在不同模式数目下的执行时间 (秒)

Algorithm\Pattern number	10	25	50	75
AC	0.47	0.57	0.64	0.67
AC_Nofailure	0.36	0.36	0.38	0.38
Opt_AC_BM	1.95	2.92	5.13	6.37
WANG	0.16	0.18	0.24	0.27
Wu-Manber	0.26	0.27	0.51	0.53

Wang 方法和 Wu-Manber 方法都是对最短字符长度敏感的,最短字符长度决定了它们可以跳过的字符的最大距离.所以,如果最短字符长度越短,则它们的性能就越差.表 3 给出了模式数目始终为 10、最短字符长度不同时,算法执行时间的比较.可以看出:AC-Nofailure 对最短模式长度不敏感;而 Wang 算法和 Wu-Manber 算法都因为最短模式长度变短,性能逐步下降,所以,它们适合于最短模式长度较大的情况.

Table 3 Execution time for the algorithms under different shortest patterns' length (s)

表 3 算法在不同最短模式长度下的执行时间 (秒)

Algorithm\Length of the shortest pattern	2	4	6	8
AC_Nofailure	0.35	0.36	0.37	0.37
WANG	0.21	0.15	0.13	0.12
Wu-Manber	0.46	0.25	0.19	0.16

5 硬件实现

在通用处理器上,软件实现的多模式匹配吞吐量通常最快只能达到 100Kbps 左右,其性能速度已不能充分满足日益增长的应用需求.网络安全设备执行如入侵检测、内容过滤等功能要求数据包达到 G 比特的扫描速度,而以 FPGA,Bloom filter,TCAM 等为代表的硬件设备是用于构建专用的高速并行处理系统的首选.

一些基于 FPGA 的技术利用片上逻辑资源将模式集合编译成状态机,用于模式的识别[6−8],但是,这样会很快耗尽存储资源,单纯的基于 FPGA 的技术不能存储大规模的模式.FPGA 外加嵌入式存储器或者 ROM 的技术则可以满足大规模模式集合的情况[9−11],而且存储器价格便宜.但是,基于存储器的设计访问延迟是潜在的性能瓶颈.基于 Bloom filter 的方法[12−14]采用了片上存储器和片外存储器相结合的方式,能够实现高速环境下对大量字符串的扫描,先利用片上的 Bloom filter 判定是否可能产生匹配,只有在可能的情况下才访问延迟高的片外存储器,从而提高了扫描效率.TCAM 由于其自身特性,通过将各个模式保存在每个表项中,每个时钟周期就可以实现对多个模式的并行查询,但是,基于 TCAM 的实现[15]对于长模式需要多步处理,而且器件成本较高、功耗大.

各种器件各有优缺点,即便是选取相同的器件,因为实现的方法不同,达到的速率也不一样.实现过程中通常采用下面的方法和策略:

1) 模式或表达式转化为自动机(NFA 或 DFA),用于 FPGA 实现[6−8].但是,因为每次比较一个字符并行性不好,只能采用多个子系统同时比较,并行的粒度很大,而且自动机状态转换受到所使用的组合逻辑的执行频率所限制,执行的速度不可能太高;
2) 预编码技术[16,17]能够更好地实现逻辑共享,使得相同的字符、字符串可以共享一个逻辑部件,减少逻辑器件的使用,但是预处理工作较为复杂;
3) Hash 方法[12,14,18]避免了逐个比较所有可能的匹配,通过过滤掉不可能的匹配,能进一步缩小比较范围,减少访问和比较次数,加速匹配.

往往在一个具体的实现中同时会用到多种方法,以期达到高速度、低成本,模式集合更新简便、快速等目的.下面分别介绍当前的结合 ROM 和存储器的 FPGA 实现方案,基于 Bloom filter 的改良的自动机实现方案和基于 TCAM 的实现方案.

5.1 结合ROM和存储器的FPGA实现

最初的实现结合了字节比较器(byte comparator)和 ROM[9],因为采用了可重写的 ROM 器件存储模式集合,从而减少了用于存储的逻辑器件,降低了器件成本.通过比较器先对字符串前缀部分进行比较,然后依据匹配的结果通过地址生成器找到 ROM 中存放相应后缀的地址,而后将后续字符串送入后缀比较器进行匹配.

进一步的改进方案增加了 hash 单元[10],这是为了过滤掉部分不可能的匹配,减少比较次数.实现由 hash 函数单元、存储器和离散字符串比较器外加一个移位控制模块构成,如图 5 所示.每个时钟周期,一部分输入字符串送入 hash 单元生成索引值,该索引指向存储器中相应的模式段,然后将获取到的模式段和输入字符串进行比较,如果产生匹配,则将该索引值送到输出,指示产生了一个相关匹配.hash 函数的输入字符串最大长度决定了该模式匹配模块可以探测的模式的最短长度,而 hash 得到的索引值的大小,决定了存储器中存放模式的表项的最大数目.

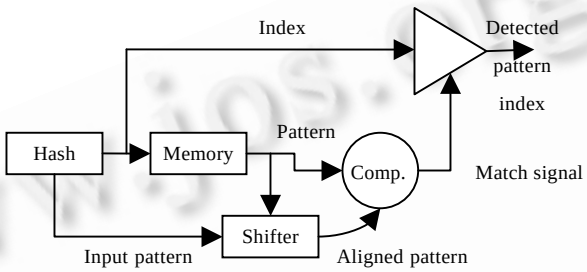

Fig.5 Pattern detection module

图 5 模式检测模块

Hash 函数的输入可以选取输入串的任意子串,用一个偏移量指明该子串的起始位置,选取方式应尽量减少 hash 冲突.由于每个模式匹配模块存储的模式数目有限,在模式数目很大的情况下,一个模块不足以解决问题,往往需要有多个模块并行操作,同一个 hash 值要送到每个匹配模块中,从而可能不只产生一个匹配.这种情况只发生在某个模式是另一个模式的前缀的情况下,可以通过优先级的设置首先找到长的匹配.

由于存在模式长度的差异,存储器宽度要能存放最大长度的模式,因而会导致存储器的浪费,因为不是每个模式都需要那么大的空间.因此,在存储器宽度一定的情况下,需要将多个模式匹配模块进行串连用于识别超出该宽度的模式.把长模式进行切分,将切分后的子模式放在串联着的几个模式匹配模块中,并添加标记字段标识它们属于哪个长模式.一个有效的切分方式是依据 A-C 算法构造出的压缩路径的模式树[1,2],既能减少存储空间,也可以减少匹配过程中潜在的模式数目.

对于每次匹配,如果长模式状态机对于输入的字符串产生了匹配,那么会输出期望的后续子模式的索引,使得下一个周期继续比较后续的输入,从而判定是否产生匹配;否则不产生任何输出.实现时可以将当前的模式集合写入 ROM 中,而将增加的模式写入存储器中,降低成本,同时也不失访问速度.对于模式集合更新,不需要重构的其他执行部件都可以采用 FPGA 或 ASIC 实现.由于对于模式规则的更新,不需要对整个设计进行逻辑重构,而仅更新存储器中的内容,所以能够有效提高更新的效率.

文献[10]给出的实验表明:该方法采用 Virtex 4 LX15 实现,对于 Snort 的 2 851 条规则共计 32 384 字节,占用存储空间 276KB,达到 2Gbps 的扫描速度.

5.2 基于Bloom filter改良的自动机实现

基本的 A-C 算法由于每次读入一个字符都需要访存获得下一个状态,因此需要频繁访问存储器,带来很大的访问延迟,而且缺乏并行性,只能采用增加多个自动机同时进行扫描的简单方式提高性能.但是,这又要求保存多个相关表的拷贝,增加了存储空间.Dharmapurikar 等人修改了基本的 A-C 算法,并引入了 Bloom filter,设计了基于硬件的匹配方案[12,14,19].匹配的步长从原来的 1 增加到 k 以提高比较的效率,每次可以扫描 k 个字符,因

此,修改基本 A-C 算法中的各个函数,构建步长为 k 的自动机,对于最后长度不足 k 的字符串也作为一个输入,对应一个状态.由于步长为 k 使得文本中模式出现的位置只有在与 k 对齐边界的情况下才能被识别出来,于是就需要有 k 个自动机,它们分别从文本开始处偏移分别为 0,1,2,…,k–1 的位置开始扫描,这样才不会漏掉任何出现的模式.而且,由于自动机的设计是基于硬件的,所以它们之间完全能够并行处理,加速扫描.依据模式集合构建的状态转换表存储以下信息:

〈当前状态和输入字符串,到达的下一个状态,产生的匹配模式,失效状态链〉.

对于每次匹配,设当前状态为 $state_j$,读入长度为 k 的文本 $x=T[i…i+k–1]$,首先找到状态转换表中当前状态为 $state_j$、输入字符串为 x 的最长前缀的表项〈$state$,$string$〉,从而获得下一个有效状态或是转到失效状态,同时输出匹配到的模式.在查找最长前缀的过程中,需要对状态转换表中状态 $state_j$ 相关的每个可能的前缀进行搜索匹配,简单的逐个搜索的方法不可行,会带来很大的延迟,因此引入 hash 思想,以〈$state_j$,$x[1…i]$〉为关键字进行 hash,状态转换表中相关表项也进行 hash,通过比较 hash 值,判断是否存在对应的表项.即便这样,最坏情况下需要对输入串 x 从 1 到 k 的每个长度的子串都进行一次 hash,再逐个比较,开销很大.为此,引入片上 Bloom filter,过滤掉不成功的搜索.依据状态表中前缀字符长度将关键字〈$state_j$,$x[1…i]$〉分类,可以分为 k 类,长度分别对应 1,2,…,k,再将同一类的关键字放入同一个 Bloom filter.在查询片外存储器搜索〈$state_j$,$x[1…i]$〉对应的表项之前,首先探测长度 i 的 Bloom filter,而 k 个 Bloom filter 可以并行查询,因此,仅一个时钟周期就可以知道这 k 类中的哪些产生了匹配,再对可能的匹配进行 hash 匹配查询,查找状态转换表获得下一个状态和相关的输出.图 6 给出了该方案的框架.采用 Bloom filter 是基于匹配发生的概率低,通常不大可能产生匹配的情况下.因此,可以省略掉一些延迟较大的查表过程,获得很高的性能.但是,计算模式集合的状态转换表需要较复杂的预处理过程.

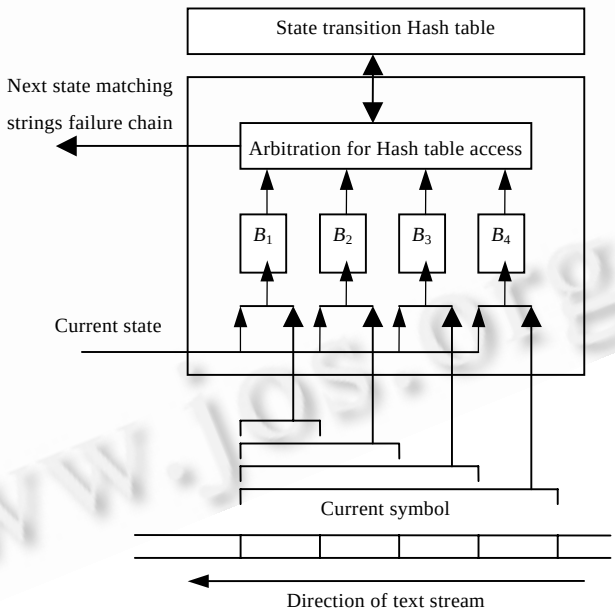

Fig.6 Bloom filter based matching framework

图 6 基于 Bloom filter 的匹配框架

文献[14]中的实验指出:使用 FPGA 器件 F=250MHz,片内存储器大小是 376Kbits,片外存储器频率是 250MHz, 64bit wide,QDRII-SRAM,采用入侵检测系统 Snort 的规则集测试,扫描速度可以达到 10Gbps.

5.3 基于TCAM的实现

TCAM(ternary content addressable memory)是一类能够进行并行查找的高速存储器,允许对位域进行 0,1 或?(忽略)三种方式的选择,它包含多个表项,每个表项可以存储一个模式串.匹配时,一个输入的字符串并行地

与 TCAM 中的每个表项的内容进行比较,输出匹配到的表项.如果 TCAM 比较的宽度为 w 字节,通常情况下,如果每个模式长度均不大于 w,那么直接将所有的模式放入 TCAM 的各个表项中,对于不足 w 的模式在其后面补“?”表示忽略.TCAM 表项的索引值是从上到下依次增加的,如果产生多个匹配,输出结果为索引值小的表项,所以,模式应该依据其长度在 TCAM 中降序排列,以保证最长匹配能够输出.查找时,首先比较文本串的前 w 个字符,如果匹配成功则输出;否则,将文本串往后移动一个字符.如此往复.

上面的情况是仅仅假设所有模式长度均小于 TCAM 的最大比较宽度 w,文献[15]给出了针对模式长度大于 w 的解决方案:将长模式进行切分,划分为多个长度为 w 的模式串,对于被切分的最后一个不足 w 的子串,将其前面子串尾部的若干个字符填充在它前面以构成 w 长的子串,假设 w=4,那么,DEFGDEF 就被切分为 DEFG 和 GDEF.通过进行多步比较,如果模式的所有子串均匹配,则说明该模式确实匹配.除了在 TCAM 中存放切分后的模式外,还需要在内存中存放多个表:组合模式表(combined pattern table)、部分命中表(partial hit list)以及匹配表(matching table).组合模式表指出每个字串是否是独立的简单模式或是哪个模式的前缀、后缀子串;部分命中表记载目前已经命中的字串的索引值以及文本中命中的位置;匹配表记录模式的前缀子串、后缀子串,前、后子串间隔的字符数和对应的、完整的模式.文本进行匹配之前,部分命中表置空,文本扫描时,若 TCAM 查找时产生了匹配,则以此按照下面 3 步处理:

1) 查组合模式表判定其是否为独立简单模式,如果是,则输出;
2) 判断是否为后缀子串,如果是,则结合部分命中表中记录的信息,依据匹配表判断是否产生了长模式匹配,如果产生了匹配则输出;如果组成了另一个前缀,那么,要将更新的匹配信息写回部分命中表,包括目前匹配出的长字串的索引和文本中匹配发生的位置;
3) 判断是否为前缀,如果是,则添加一个表项到部分匹配表,记录匹配字串的索引值以及文本中匹配发生的位置.

设 TCAM 的比较宽度为 w 个字符,每个模式长度为 m_i,那么,所有模式的存储空间需要 $w^*\sum\left\lceil\frac{m_i}{w}\right\rceil$,前缀和后缀索引数目为 $\sum\left(\left\lceil\frac{m_i}{w}\right\rceil-1\right)$.对于每次 TCAM 查找,如果命中,则需要进一步访问通用存储器,对相关的表进行读写,而访问通用存储器和 TCAM 可以以流水方式进行.通常情况下,由于命中的概率很小,所以访存的时间可以忽略不计,TCAM 查询时间决定了模式匹配的速度.假设在 TCAM 的查询时间为 4ns 的情况下,文本的扫描速度为 8 bits/4ns=2Gbps.TCAM 虽然执行的速度快,实现较简单,有利于短模式的匹配,但是,每个模式的存储相对独立,逻辑器件的共享性不好,而且价格昂贵,功耗较大.

6 展 望

多模式匹配发展到今天已经有 20 多年的历史了,从最初的对文本的检索,到现在对高速网络流量中内容的过滤、入侵代码的识别.单纯的软件实现,在速度上已经不能够满足日益增长的需求,FPGA 或 ASIC 等专用器件的使用能够有效提高匹配速度,但同时,设计相应的模块也会带来一定的复杂性.

在软件的实现上主要考虑的是预处理的复杂度,执行过程中的时间、空间复杂度以及实现上的难易程度.针对这些问题,提出了不同的改进方法:

1) 使用 hash 减少不可能匹配,增加每次比较的字符数目可以加快匹配进程:Wu-Manber[4]算法对每个字符子串采用了 hash 计算,过滤掉不可能匹配的字符串,有效地避免了字符的逐一匹配,对于模式在文本中出现概率不大的情况,可显著加快匹配速度;

2) 采用启发式方法,引入跳跃加速比较:Wu-Manber[4]算法通过预计算得到的 SHIFT 表就是利用了模式自身的规律,计算出在读到当前子串后可以向前跳过的位数,加速了处理流程;Wang 方法[3]逆向构建自动机,对于模式采取从右向左进行比较,利用“坏字符”启发式规则,跳过了不可能的匹配加速处理;

3) 通过状态的合并与删减等操作可以减少存储空间的需求,同时也能加速匹配:去掉了 failure 函数的 A-C

算法[1]通过合并 goto 函数和 failure 函数,减少了失效状况下转移的次数,提高了匹配速度;基于位图压缩的 A-C 算法[2]仅保存可能到达的下一状态信息,删减了保存不可能到达的状态的存储空间,有效地减少了存储空间的需求.

这些方面相互影响和制约:时间、空间上的优化往往增加预处理的复杂度;时间上优越的方法通常需要较大的存储空间;空间优化的方法会增加计算复杂度制约执行的时间.因此,很难有一种各方面都很优异的算法,而往往是各方面的折衷.但是可以针对具体需求,如模式长度、数目需求特定的情况下,给出合理的改进,找出适合于具体应用的匹配方法.

软件实现虽然灵活,但是不能满足高速处理的要求,入侵检测和内容过滤等都对模式匹配提出了很高的要求,要设计出高速、能够处理长模式的、同时易于更新模式集合的模式匹配模块.因此,当前的研究实现主要以 FPGA,TCAM 等高速可并行的硬件设备为基础.对于在这些硬件上的实现方法都是当前研究的热点,其目的就是围绕成本、速度、模式的规模和实现的复杂度等关键点出发,设计出合理的方案.而通常的改进主要是通过使用辅助硬件单元和使用优化策略提高整体性能.

1) 在辅助硬件单元方面,ROM 和存储器的使用能满足模式集合规模很大的需求,同时使得模式集合的更新更快捷[9−11];Bloom filter 的使用能过滤掉不可能的匹配,减少匹配的次数[12−14].因此,选取合适的硬件辅助器件用于模式匹配的 FPGA 实现中,有利于提高匹配的性能.

2) 在优化策略方面,hash 的使用[15,16,20]简化了比较过程,同时也能起到过滤不可能模式的作用,加速了比较进程;预编码的使用[8,21]能够共享逻辑器件,节约了资源,降低了成本,在其他情况不变的情况下,可以容纳更多的模式;软件与硬件结合的方式[12−14,19]能够利用软件的灵活性和硬件的高速性;采用合适的优化方式,对提高整体的性能有很大的帮助.

随着高速处理要求的日益增加,可以预计,今后多模式匹配的应用将会更多地以硬件方式实现为主,但为了兼顾灵活,也会相应地引入一些辅助的软件策略.目前,在普通 FPGA 器件上,以 Snort 规则集作为模式集合的实现,其吞吐量一般不超过 10G,还存在一定的上升空间,无论是在辅助器件还是在优化策略方面,都可以提出新的改进,以期达到更高的吞吐量.

致谢 感谢中国科学院计算机网络信息中心网络技术与应用研究室的各位同事、同学的关心与支持,尤其感谢网络体系结构研究组的所有成员,与诸位的讨论使我们受益匪浅.

References:

[1] Aho AV, Corasick MJ. Efficient string matching: An aid to bibliographic search. Communications of the ACM, 1975,18(6): 333−340.

[2] Tuck N, Sherwood T, Calder B, Varghese G. Deterministic memory efficient string matching algorithms for intrusion detection. In: Li VOK, ed. Proc. of the IEEE Infocom 2004. Piscataway: IEEE, 2004. 333−340.

[3] Wang YC, Shen Z, Xu YZ. Improved algorithms for matching multiple patterns. Journal of Computer Research and Development, 2002,39(1):55−60 (in Chinese with English abstract).

[4] Wu S, Manber U. A fast algorithm for multi-pattern searching. Technical Report, TR 94-17, University of Arizona at Tuscon, 1994.

[5] Wu S, Manber U. Agrep—A fast approximate pattern matching tool. In: Proc. of the USENIX Winter Technical Conf. USENIX, 1992. 153−162.

[6] Franklin F, Carver D, Hutchings B. Assisting network intrusion detection with reconfigurable hardware. In: Pocek KL, ed. Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines. Los Alamitos: IEEE Computer Society, 2002. 111−120.

[7] Moscola J, Lockwood J, Loui RP, Pachos M. Implementation of a content-scanning module for an Internet firewall. In: Pocek KL, ed. Proc. of the 11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines. Los Alamitos: IEEE Computer Society, 2003. 31−38.

[8] Sidhu R, Prasanna VK. Fast regular expression matching using FPGAs. In: Pocek KL, ed. Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines. Los Alamitos: IEEE Computer Society, 2001. 227−238.

[9] Cho YH, Mangione-Smith WH. Deep packet filter with dedicated logic and read only memories. In: Pocek KL, ed. Proc. of the 12th Annual IEEE Symp. on Field-Programmable Custom Computing Machines. Los Alamitos: IEEE Computer Society, 2004. 125−134.

[10] Cho YH, Mangione-Smith WH. Fast reconfiguring deep packet filter for 1+gigabit network. In: Pocek KL, ed. Proc. of the 13th Annual IEEE Symp. on Field-Programmable Custom Computing Machines. Los Alamitos: IEEE Computer Society, 2005. 215−224.

[11] Cho YH, Mangione-Smith WH. A pattern matching coprocessor for network security. In: Joyner WH, ed. Proc. of the 42nd Annual Conf. on Design Automation. New York: ACM Press, 2005. 234 −239.

[12] Dharmapurikar S, Krishnamurthy P, Spoull T, Lockwood J. Deep packet inspection using bloom filters. In: Lockwood J, ed. Proc. of the Hot Interconnects. Washington: IEEE Computer Society, 2003. 52−61.

[13] Dharmapurikar S, Attig M, Lockwood J. Design and implementation of a string matching system for network intrusion detection using FPGA-based bloom filters. Technical Report, Saint Louis: Washington University, 2004.

[14] Dharmapurikar S, Lockwood J. Fast and scalable pattern matching for content filtering. In: Berenbaum A, ed. Proc. of the 2005 Symp. on Architecture for Networking and Communications Systems. New York: ACM Press, 2005. 183−192.

[15] Yu F, Katz RH, Lakshman TV. Gigabit rate packet pattern-matching using TCAM. In: Koenig H, ed. Proc. of the 12th IEEE Int'l Conf. on Network Protocols (ICNP 2004). Washington: IEEE Computer Society, 2004. 174−183.

[16] Sourdis I, Pnevmatikatos D. Pre-Decoded CAMs for efficient and high-speed NIDS pattern matching. In: Pocek KL, ed. Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines. Los Alamitos: IEEE Computer Society, 2004. 258−267.

[17] Cho YH, Mangione-Smith WH. Deep packet filter with dedicated logic and read only memories. In: Pocek KL, ed. Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines. Los Alamitos: IEEE Computer Society, 2004. 125−134.

[18] Papadopoulos G, Pnevmatikatos D. Hashing+Memory=Low cost, exact pattern matching. In: Rissa T, ed. Proc. of the 15th Int'l Conf. on Field Programmable Logic and Applications. Piscataway: IEEE, 2005. 39−44.

[19] Attig M, Dharmapurikar S, Lockwood JW. Implementation results of bloom filters for string matching. In: Pocek KL, ed. Proc. of the 12th Annual IEEE Symp. on Field-Programmable Custom Computing Machines. Los Alamitos: IEEE Computer Society, 2004. 322−323.

[20] Sourdis I, Pnevmatikatos D. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. In: Cheung PYK, ed. Proc. of the 13th Int'l Conf. on Field-Programmable Logic and Applications. LNCS 2778, Berlin: Springer-Verlag, 2003. 880−889.

[21] Gokhale M, Dubois D, Dubois A, Boorman M, Poole S, Hogsett Granidt V. Towards gigabit rate network intrusion detection technology. In: Glesner M, ed. Proc. of the 12th Int'l Conf. on Field-Programmable Logic and Applications. LNCS 2438, Berlin: Springer-Verlag, 2002. 404−413.

附中文参考文献:

[3] 王永成,沈州,许一震.改进的多模式匹配算法.计算机研究与发展,2002,39(1):55−60.

李伟男(1981－),男,湖北武汉人,硕士生,主要研究领域为网络安全,网络处理器的应用.

葛敬国(1973－),男,博士,助理研究员,主要研究领域为互联网体系结构.

鄂跃鹏(1976－),男,硕士生,主要研究领域为网络协议,嵌入式.

钱华林(1940－),男,研究员,博士生导师,主要研究领域为下一代网络体系结构.